(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
20 November 2003 (20.11.2003)

PCT

(10) International Publication Number
WO 03/096544 A1

(51) International Patent Classification[7]: H03M 7/30

(21) International Application Number: PCT/GB03/00388

(22) International Filing Date: 30 January 2003 (30.01.2003)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
0210602.9 9 May 2002 (09.05.2002) GB

(71) Applicant *(for all designated States except US)*: **INTERNATIONAL BUSINESS MACHINES CORPORATION** [US/US]; New Orchard Road, Armonk, NY 10504 (US).

(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: **COCKBURN, Gordon** [GB/GB]; 56 Shirley Avenue, Southampton, Hampshire SO15 5NJ (GB). **HAWES, Adrian, John** [GB/GB]; 9 Beechwood Close, Chandlers Ford, Eastleigh, Hampshire SO53 5PB (GB).

(74) Agent: **BURT, Roger, James**; IBM United Kingdom Limited, Intellectual Property Law, Hursley Park, Winchester, Hampshire SO21 2JN (GB).

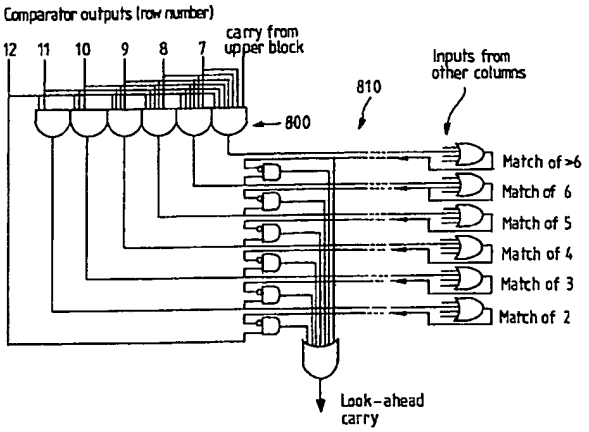
(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SD, SE, SG, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IT, LU, MC, NL, PT, SE, SI, SK, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *with international search report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: CIRCUIT AND METHOD FOR USE IN DATA COMPRESSION



(57) Abstract: A circuit (640) and method for use in a comparison unit (400) of a comparison matrix (170) for LZ1 compression of a data string by comparing bytes held in an input buffer (140) with bytes held in a history buffer (110, 120), the method comprising: providing a plurality of logic gate stages (720, 730, 740, 750) coupled in series, each of the plurality of logic gate stages producing a carry value which is passed to one of the output of the unit and another of the plurality of logic gate stages, the product of the number stages in the plurality of logic gate stages and the number of logic gates in each of the plurality of logic gate stages being less than the number of logic gates required for an equivalent circuit having a single logic circuit stage. The carry value represents the number of consecutive matches detected by the series of logic gate stages. Delay resulting from using the plurality of logic gate stages may be reduced by using a carry look-ahead value (770). Compression efficiency may be improved by comparing the carry value produced at the unit's output with an earlier-generated carry look-ahead value, to determine whether the carry value could be improved.

WO 03/096544 A1

METHOD AND APPARATUS FOR PARALLEL DATA COMPRESSION ACCORDING TO THE LZ77 ALGORITHM

Field of the Invention

This invention relates to data compression, and particularly to 'LZ1' data compression.

Background of the Invention

The Lempel-Ziv algorithms are well known in the field of data compression. In particular, the "history buffer" version, known as LZ1, has become particularly popular in hardware implementations wherever lossless compression of coded data is required, since its relatively modest buffer requirements and predictable performance make it a good fit for most technologies.

The LZ1 algorithm works by examining the input string of characters and keeping a record of the characters it has seen. Then, when a string appears that has occurred before in recent history, it is replaced in the output string by a "token": a code indicating where in the past the string has occurred and for how long. Both the compressor and decompressor must use a "history buffer" of a defined length, but otherwise no more information need be passed between them.

Characters that have not been seen before in a worthwhile string are coded as a "literal". This amounts to an expansion of the number of bits required, but in most types of data the opportunities for token substitution (and hence compression) outweigh the incompressible data, so overall compression is achieved. Typical compression ratios range from 2:1 to around 10:1.

Some variations of the basic LZ1 algorithm have emerged over the years, but improvements have been incremental.

As the LZ1 algorithm works on units of a byte, traditional hardware implementations consider just one byte at a time when compressing the input stream. As each byte is input, the "history buffer" is scanned - using, for example, a Content-Addressable-Memory (CAM) - for all occurrences of the byte. As a single byte is not considered an efficient candidate for string replacement, any matches found must be supplemented by consecutive matches before token substitution takes place.

Each subsequent byte that is input is also sought in the history buffer, but the only matches reported are those following existing matches. Finally, the string match may terminate (when no more matches found that adjoin known matches) and the surviving "string match" is coded for substitution. The longer the match, the greater the saving in bits.

A simple implementation of the LZ1 algorithm which processes one byte per clock cycle is limited to some 100-200 MB/s at typical clock rates for current ASIC (application specific integrated circuit) technology. However, this may be insufficient for some applications (such as, for example, memory compression, optical networks and RAID disk arrays) which require high bandwidth for a single data stream. To increase performance, that is, the number of bytes that may be compressed per second, either the "cycle time" (the time taken to input the byte and find all matches) must be reduced, or the CAM be modified to search for more than one byte at a time. Because of the difficulty of designing multiple-input CAMs, performance improvements have usually concentrated on shortening the access time of the CAM, and hence the cycle time. But of course, the two improvements are not mutually exclusive; a multi-byte CAM will gain performance over and above any reduction in cycle time.

In GB patent application (the content of which is hereby incorporated herein by reference) entitled "METHOD AND ARRANGEMENT FOR DATA COMPRESSION", having applicant's reference GB9-2001-0056, and filed by the same applicant contemporaneously with the present application, there is described a method and arrangement for LZ1 compression of a data string by holding in an input buffer a first sequence of bytes of the data string; holding in a history buffer a second sequence of bytes of the data string; comparing, in matrix comparison means coupled to the input buffer and the history buffer and having a plurality of rows and columns of comparison units, bytes held in the input buffer with bytes held in the history buffer, bytes of the history buffer being coupled to diagonally displaced comparison units in the matrix comparison means; detecting in each of the rows the column in which a largest number of consecutive byte matches has occurred at the comparison unit in that row and preceding comparison units in the same column; and encoding as a token a sequence of matched bytes detected in the step of detecting.

However, although this technique will extend to any number of bytes per cycle, the number of logic gates consumed in this implementation increases with the square of the number of bytes, so designs beyond about 4 bytes per cycle are costly.

A need therefore exists for a method and arrangement for data compression wherein the above-mentioned disadvantage(s) may be alleviated.

Statement of Invention

In accordance with a first aspect of the present invention there is provided a circuit, for use in a comparison matrix for LZ1 compression of a data string, as claimed in claim 1.

In accordance with a second aspect of the present invention there is provided an arrangement, for LZ1 compression of a data string, as claimed in claim 7.

In accordance with a third aspect of the present invention there is provided a method, for use in a comparison matrix for LZ1 compression of a data string, as claimed in claim 8.

In accordance with a fourth aspect of the present invention there is provided a method, for LZ1 compression of a data string, as claimed in claim 14.

Brief Description of the Drawings

One method and arrangement for data compression incorporating the present invention will now be described, by way of example only, with reference to the accompanying drawing(s), in which:

FIG. 1 shows a block schematic diagram of a comparison matrix used in a compression arrangement utilising the present invention;

FIG. 2 shows a block schematic diagram illustrating in detail a comparison unit of the compression arrangement of FIG. 1;

FIG. 3 shows a schematic diagram illustrating compression operation in the compression arrangement of FIG. 1;

FIG. 4 shows a block schematic diagram illustrating in detail a particular implementation of a comparison unit matrix used in the compression arrangement of FIG. 1;

FIG. 5 shows a schematic diagram illustrating reduction in gate count achieved by use of the present invention;

FIG. 6 shows a block schematic diagram illustrating an example of 4-way breakdown which may be used in the present invention;

FIG. 7 shows a block schematic diagram illustrating delay reduction which may be used in the present invention;

FIG. 8 shows a block schematic diagram illustrating generation of a look-ahead carry which may be used in the present invention;

FIG. 9 shows a block schematic diagram illustrating an example of a 'carry look-ahead pessimism' problem which may arise in the present invention; and

FIG. 10 shows a signal timing diagram block illustrating restoration of compression efficiency which may be achieved in the present invention.

Description of Preferred Embodiment(s)

The following preferred embodiment is described in the context of an LZ1 variant attributed to IBM and known as "IBMLZ1", but it will be understood that the technique presented is applicable to all versions of the LZ1 algorithm. This technique is expandable to any number of bytes per cycle, or any length of history buffer, but it will be described using a 12-byte-per-cycle design with a 512-byte history buffer.

Referring now to FIG. 1, a compression arrangement 100 includes two groups (L1 and L2) of 512-byte latches 110 and 120, a group of 512 carry latches 130, a 12-byte input buffer 140, a MAX Function/Priority Encoder 150, a token encoder 160, and a 512-by-12 matrix array 170 of comparison units 200 (which will be described in greater detail below).

The L2 latches 120 are coupled respectively to 512 comparison units in the first row of the matrix 170, and to comparison units diagonally displaced successively by one unit to the right in each lower row of the matrix as will be explained in more detail below. The L1 latches 110 and associated carry latches 130 are coupled respectively to the 512 comparison units in the last row of the matrix 170. The 12 bytes of the input buffer 140 are coupled respectively to the 12 rows of comparison units in the leftmost column of the matrix 170. The MAX Function/Priority Encoder 150 and token encoder 160 are coupled to the 12 rows of comparison units in the matrix 170.

The 12 input bytes have to be compared with the entire history buffer, in the search for matches. However, some of the input bytes themselves constitute part of the "history". A 12-byte input buffer must assume that each byte is in chronological order, even though they all enter in one cycle. Therefore one end of the input buffer is considered "most recent", and the other "least recent", as if the "least recent" bytes entered the process first. Each byte must consider those in the input buffer that are "less recent" to be part of the history, and be compared accordingly.

The manner in which the input bytes are compared, both with the bytes in the history buffer and the less recent bytes in the input buffer, is shown in FIG. 1. Considering the input buffer 140 on the left of the diagram, if the processing were the conventional type - one byte at a time - then the top byte would be the first in and the bottom byte the last; however, in this implementation the bytes are all input at the same time. As can be seen, the outputs of the input buffer - all 12 input bytes - are connected to the inputs of all comparison units 200 in each row of the matrix 170. In each clock cycle the contents of the history buffer - all 512 bytes of it - are fed down for comparison with the first (least recent) byte of the input buffer, and then diagonally down and across for comparison with the most recent byte of the input buffer.

It will be understood that, as every byte of the history buffer must be available at once, conventional RAM cannot be used for the history buffer. In this implementation the history buffer is constructed using registers, arranged as level sensitive scan design (LSSD) L1-L2 pairs. At the end of each clock cycle the 12 input bytes are shifted into the history buffer, the old contents of the history buffer are shifted along (to the right as shown in FIG. 1) by 12 bytes, and the oldest 12 bytes are discarded.

The comparison units 200, represented by clipped squares in FIG. 1, are a fundamental element of this design. An expanded diagram of a comparison unit is shown in FIG. 2. It is the job of each block to compare the values of the two input bytes, tally up the count of matched bytes, and report a winning match to control logic.

A comparison unit 200 in the matrix 170 includes a byte comparator 210 arranged to receive for comparison a byte value from the input buffer byte position for that row and a history buffer byte value passed from a unit diagonally left and above. An incrementer 220 is arranged to receive

and increment by '1' a 'count in' value from a unit directly above in the same column of the matrix. A selector 230 is arranged to receive the incremented count value and a '0' value and to select between these in dependence on the output of the comparator 210. If the comparator 210 indicates a match, the selector 230 outputs the incremented count value; otherwise it outputs a '0' value. The output of the selector is passed as a 'count out' value to a unit directly below in the same column; the selector output is also passed to MF/PE for the same row of the matrix. As shown by the thick dashed lines, the byte values input to the selector 210 are passed to a unit directly to the right in the same row and to a unit diagonally below and right.

FIG. 2 shows that in addition to the byte to be compared, the unit 200 takes as input the "count" from the unit above, which indicates the length of the string seen thus far, and the byte for the same row of the input buffer 140. If the two input bytes match, then the comparison unit will increment that count, and pass the new count to the unit below it. If the two bytes do not match then the output count will be set to zero, regardless of the input count value.

The value of this count is also output from the right-hand side of each comparison unit, and is fed to the "MAX Function/Priority Encoder" (MF/PE) logic 150 at the end of the row. There is one of these MF/PE units for each of the twelve rows of the compressor 100. The function of the MF/PE is to decide which comparison unit(s) 200 of the 512 in that row reported the longest string (i.e., the largest count - the MAX function), and to encode the position within the row. If more than one column produces the same large count then the MF/PE encodes (arbitrarily) the left-most value (the priority encoding function). However, it may be noted that the value produced by the MF/PE is not necessarily the string that will be encoded, as the string may continue beyond that row.

String matches that are still alive in row 12 (the last row of the matrix 170) may continue into the next cycle. The carry latches 130 at the bottom of FIG. 1 store the position of any surviving strings from this row. (The length of that string - the "count" - is stored in a separate single register, not shown.) The carry is fed into the "count input" to the first row of comparison units in the next cycle. It may be noted that there is a limit to the string length that can be encoded by the LZ1 algorithm, imposed by the number of bits in the token. (In IBMLZ1 the limit is 271 characters.) When the maximum number is reached a token is emitted and the string must start from zero. It will be appreciated that

the token encoder 160 operates in the same manner known in the prior art and its structure and function need not be described in any further detail.

The largest string values for each row (reported by the MF/PE 150) and their encoded positions are fed to the Token Encoder (TE) 160. The TE examines the reported lengths for each row, and decides where strings can be encoded for this batch of 12 input bytes. Where strings are to be encoded, the TE uses the positions reported by the MF/PE as part of the token, along with the string length. Note that the length may rise to more than 12, when a long string spans more than one cycle. When this happens, the count is accumulated in the TE, ready for coding when the string terminates.

If no strings are found (noting that a match of just one byte is not worth encoding) or if some of the 12 bytes cannot be matched, then the TE must output literals. For this the TE takes data directly from the input buffer 140.

FIG. 3 shows a snapshot of a compression operation 300. The previous sentence is used as input, and for simplicity only 5 input bytes and 27 history buffer bytes are shown. The filled circles (at columns 310, 320, 330, 340 and 350) indicate where a match is detected; a useful string match can be seen at column 320 in the current input bytes "ion". It is the position of column 320 in the row that will be forwarded for encoding.

A possible implementation 400 for the comparison unit is shown in FIG. 4. The comparison unit 400 (which as illustrated is a unit of row 3 of the matrix 170) has a comparator 410 which receives the two byte values to be compared as described above. Three AND gates 422, 424 and 426 each have one of their inputs connected to receive the output of the comparator 410, and have their other inputs connected to receive respective ones of three bit lines (carrying a 3-bit 'input count' value) from a comparison unit directly above in the same column. The outputs of the AND gates 422, 424 and 426, together with the output of the comparator 410, (carrying a 4-bit 'output count' value) are connected to a comparison unit directly below in the same column. The leftmost three of the 'output count' bit lines are connected respectively to inputs of three AND gates 432, 434 and 436. The outputs of the AND gates 432, 434 and 436, together with the output of the AND gate 426, are connected to inputs of a 4-input OR gate 440.

The output of the OR gate 440 (together with outputs of the other comparison units 400 in row 3 of the matrix 170) are connected, within an MF/PE 500, to inputs of a 512-input priority encoder 510. Also within the MF/PE 500, the outputs of the AND gates 422, 424 and 426 are connected (together with outputs of AND gates of other comparison units 400 in row 3 of the matrix 170) to respective inputs of 512-input OR gates 522, 524 and 526. the outputs of the OR gates 522, 524 and 526 are connected invertedly to inputs of the AND gates 432, 434 and 436 in each of the comparison units 400 in row 3 of the matrix 170.

The comparator 410 is the same as the comparator 210 in the comparison unit 200 described above, but in the comparison unit 400 the "count" is maintained by an N-bit vector. The bits of this vector are numbered 1 to N, and a count of "n" is represented by bits 1 to n being '1'. All other bits in the vector are '0'. Thus, a count of zero is shown by all bits being zero. This is a useful method of counting in this design because:

1. The number of bits required, N, need only be as large as the row number (row 1 needs only 1 bit, row 12 needs 12 bits),

2. The "count" is easily incremented, merely shifting to the right with a '1' fill, and

3. A MAX function is easily implemented, by ORing the respective bits of all the vectors together.

With a small amount of extra logic (in the form of the AND gates 432, 434 and 436 and the OR gate 440) in the comparison unit, the priority encoder is made simple also.

In operation of the comparison unit 400 of FIG. 4, this works as follows. The input count is represented by a 3-bit vector, which can indicate 4 values:

Vector	Indicated Value	Indicated Match
000	zero	
100	one	match in this column in row 2
110	two	match in this column in rows 1 and 2
111	more than two	match in this column in rows 1 and 2, and a carry

If the comparator 410 detects a match in this column in this row (row 3), it will increment the count, producing a 4-bit vector 'output count' from the bottom of the unit. The incrementation will be achieved by effectively shifting the input vector right by one bit, adding a '1' at the left. If there is no match here, the AND gates 422, 424 and 426 are all turned off and the 'output count' collapses to zero.

A modified version of the 4-bit count is output to logic circuitry 510, 522, 524 and 526 in the MF/PE 500 at the end of the row, also shown in FIG. 4. The three 512-input OR gates 522, 524 and 526 decide the maximum count for this row (the low-order bit is not used as it represents a count of only 1 byte). This maximum value is used to disqualify all counts smaller than the winning count, by means of AND gates 432, 434 and 436 in the comparison units 400 of the columns that do not contribute this maximum count. Those comparison units that show the maximum counts declare their candidacy on the encoder input, and the priority encoder codes the position of the leftmost of them. The "win" outputs of OR gates 440 in the comparison units of the bottom row comprise the 512 "carry" bits to be stored for the next cycle.

Although the above description has shown all operations happening in a single cycle, the design does not preclude some elements of pipelining. The critical factor is that the carry for each row must be successfully stored in one cycle, ready for inclusion in the next cycle's calculations.

The critical timing path - producing a valid carry for the next cycle - consists in this case of up to 12 AND gates (from a string of 12 matches), through a 512-input OR gate (which probably has several cascaded gates) and then through two more AND gates for the carry.

Thus, a total of some 20 gate delays determines the maximum operating frequency for this design. The 12 AND gate delays may be reduced by using look-ahead techniques, although this adds to the gate count.

It can be understood that comparison units 400 in the lower rows of the matrix 170 have proportionally more gates, and so it can be understood that the total number of gates increases with the square of the number of bytes processed per cycle.

Reducing Gate count

Since the total number of gates increases with the square of the number (N) of bytes processed per cycle (in accordance with the formula

N(N+1)/2), the total number of gates may become impractical for larger numbers of bytes (e.g., greater than 4) processed per cycle. In order to reduce this impracticality, a further technique may be used to reduce the total number of gates.

Referring now to FIG. 5A, each column of the comparison matrix 170 may be considered as a 'triangle' (600) of gates, with the smaller number at the top of the triangle. The output at the bottom of the triangle is the 'carry' which indicates the longest surviving string(s), and which is stored for the next cycle. The area of the triangle is proportional to the number of gates used.

Referring now to FIG. 5B, greater efficiency may be achieved by breaking each triangle into smaller triangles (610, 620, 630), with a correspondingly smaller total area in the arrangement 640. Breaking the single triangle of FIG. 5A into the three triangles 610, 620 and 630 of FIG. 5B results in a reduction from 78 to 30 gates.

Each smaller triangle must produce its own 'carry' which is input to the next triangle below. This carry is true on all the columns that signal the longest current string, and so requires feedback from the Max Function / Priority Encoder (MF/PE logic). Each carry generation therefore incurs extra delay, and the sum total of all these delays limits the number of triangles that the design can be broken down into, and hence the savings that can be made.

A more detailed example is shown in FIG. 6. The original design (corresponding to the triangular gate arrangement shown in FIG 5A), shown on the left and bottom of the diagram as FIG. 6A, is a 12-byte-per-cycle one, using 12*(12+1)/2 or 78 gates for the string accumulation logic. The time through to the carry out is approximately (12+3)=13 gate delays. The reduced design (of which one quarter is shown in the dashed-line box at the top right of the diagram as FIG. 6B) breaks this into 4 iterations (as compared with the arrangement of FIG. 5B which has 3 iterations) of a 3-byte-per-cycle comparator, costing 4*3*(3+1)/2 or 24 gates, with a through time of 4*(3+3)=24 gate delays.

Reducing Delay

Although in the arrangement of FIG. 6B the gate count has been reduced significantly, the consequent increased delay may be too great for the required cycle time. To counteract this, some pipelining may be employed.

In FIG. 7A is shown an abbreviated form of FIG. 6B, with a latch 710 and 4 stages or triangles of gates 720, 730, 740 and 750. FIG. 7B shows how the design of FIG. 6B may be further enhanced to reduce delay per cycle. In the arrangement of FIG. 7B the carry chain of FIG. 7A has been split in two, and the result of the first half carry *(B)* from stage 730 is latched (in latch 760) ready for the second cycle *(D)*. On the second cycle the latched carry is propagated down the remainder of the chain, while the first half processes the carry for the next cycle. Thus the required cycle time has been cut in half.

There is a complication, however, in that the first half of the carry chain requires as input *(A)* the *complete* carry for the previous cycle. It requires the same value as would be produced from the arrangement of FIG. 7A. But the previous cycle only produced the carry from the first half of the chain, as the cycle is too short to produce the whole carry.

To solve this, the first half of the carry chain must produce a *carry look-ahead (C)* that generates the carry for the full chain. Note that it is not necessary to do the full processing of the column data (that will be accomplished in the second cycle by the logic to the right of the latch 760). It is merely necessary to provide the *carry* (the column or columns that have the largest surviving string). This carry may be generated in fewer gates than the full processing provided by the second set of gates. This carry look-ahead *(C)* is produced at stage 770.

Carry Look-Ahead

FIG. 8 shows how this look-ahead carry is generated. The outputs of the six lower comparators (three each from stages 740 and 750) are used along with the carry output from the upper block (signal 'B' in FIG. 7B). Logic 810 similar to the carry generator at the bottom of FIG. 6A determines the best surviving string *in the twelfth row*. This is a suitable carry for use in the next cycle.

Carry look-ahead pessimism

However, while the carry generated by the look-ahead logic of FIG. 8 is a valid one, it may not give the optimum compression. Because it concentrates on row 12 (which it needs to in order to meet timing constraints) it may miss some other string candidates.

FIG. 9 shows an example of how this can happen. The diagram shows three strings (910, 920 and 930) occurring in the bottom six rows. The leftmost string (910) of length 4 has won the look-ahead contest, as it is longer than the rightmost string (930) of length 3. But the middle string (920) of length 3 (not seen by the look-ahead) would have generated the carry from the upper block of three, and eclipsed the leftmost string (910) at this point. With no carry from the upper block, the leftmost string (910) would have begun on the next line, and its length would be reduced to 3. Thus both strings (910 and 930) surviving on the last line would have generated a carry; the rightmost (930)could easily provide a longer match eventually.

The consequence of this 'pessimism' (in assuming that the rightmost string 930 would not be the eventual winner) is that fewer carries are stored, which in turn means that strings that could have gone on to efficient lengths in the next cycle are not spotted. This will not produce an error, but may slightly reduce the compression efficiency of the encoder.

Compression efficiency may be restored, however, by sacrificing some of the performance. The look-ahead carry generator will always produce the same number as or fewer carries than the full decode. The result of the full decode is known on the *second* cycle: normally too late for the full speed operation. By comparing the latched values of the look-ahead carry with the full carry output from the second cycle, it may be determined whether the first cycle carry was inefficient. If it was, the cycle may be repeated, with the new carry settings.

FIG. 10 shows an example of how this would operate. The diagram shows the signals at the five points *A*, *B*, C, *D* and *E* from FIG. 7B, which are the generated carries at various stages. The numbers on the signals indicate the input cycle for which that particular carry was generated. The question mark shows that the carry generated may not be correct, i.e. may be too pessimistic.

In the first cycle, the latched look-ahead was generated during input number 1. During this cycle, the top block generates a carry (2 at *B)*, and later the look-ahead carry (2? at C) is generated. The quality of the look-ahead is unsure at this point, hence the '?'. Meanwhile, *(D)* has the latched carry from input number 1, and generates the carry for the bottom half of the data at *(E)*.

In the second cycle, *(A)* has the latched uncertain look-ahead carry. The carries generated at both *(B)* and (C) will be uncertain (shown as 3?), because they both depend on *(A)*. *(D)* and *(E)*, however, show good carries, because they have been derived from the full chain, albeit one cycle delayed.

At the end of the second cycle (indicated at 1000') the latched look-ahead carry at *(A)* is compared with the true carry from *(E)*. As in this example it is assumed they are the same, the latched look-ahead carry (2 on *A)* was good, and so the carry derived from it (3 on *B)* was also good. Note that the look-ahead generated in *this* cycle (3? on C) cannot be trusted yet (the carry value might possibly be improved), and so its latched version *(A)* on the next cycle still shows a '?'.

The third cycle is similar, except it is assumed that the comparison between *(A)* and *(E)* at the cycle's end (indicated at 1000'') fails. The latched look-ahead carry (3?) was too pessimistic. So the results derived from that (the two 4?'s) are discarded and *(A)* is latched from *(E)* rather than (C). The cycle is repeated - now with a known good carry - and new carries are generated.

It can be seen that the extra cycle added on the rare occasions when the look-ahead carry is too pessimistic (such as the case in FIG. 9) will restore full efficiency. It should be remembered that the compression operation *without* this check will still be correct (will decompress correctly), but may not show as large a compression ratio. Accordingly, the use of the above technique should be selectable, depending on whether speed or high compression ratio is desired.

Summary

In summary, it will be understood that the invention provides an improvement to the earlier LZ1 multi-byte compression engine, that dramatically reduces the number of gates required for a given number of bytes per cycle. Pipelining can be employed to compensate for the extra delays, which necessitates a *look-ahead carry* generator. Deficiencies in the look-ahead carry can be corrected with a small loss in performance.

It will be also be appreciated that the arrangement and method described above will typically be carried out in hardware, for example in the form of an integrated circuit (not shown) such as an ASIC (Application Specific Integrated Circuit).

It will be understood that the data compression technique described above may be used in a wide range of storage and communication applications.

CLAIMS

1. A circuit (640) for use in a comparison matrix (170) for LZ1 compression of a data string by comparing bytes held in an input buffer (140) with bytes held in a history buffer (110, 120), the circuit comprising:

a plurality of logic gate stages (720, 730, 740, 750) coupled in series, each of the plurality of logic gate stages being arranged to produce a carry value for passing to one of the output of the unit and another of the plurality of logic gate stages, the product of the number stages in the plurality of logic gate stages and the number logic gates in each of the plurality of logic gate stages being less than the number of logic gates required for an equivalent circuit having a single logic circuit stage.

2. The circuit of claim 1 further comprising an input latch (710) coupled to the input of the circuit for receiving a carry value from another carry-value-producing circuit.

3. The circuit of claim 1 or 2 further comprising means (770) coupled to a predetermined number of the plurality of logic gate stages for producing a look-ahead carry value.

4. The circuit of claim 3 further comprising an intermediate latch (760) coupled to receive the carry value from a logic gate stage coupled to the means (770) for producing a look-ahead carry value.

5. The circuit of claim 3 or 4 wherein the means (770) for producing a look-ahead carry value comprises logic means (810) coupled to outputs of comparators via logic gates (800) and arranged to be coupled to comparison units of other columns of the comparison matrix.

6. The circuit of claim 3, 4 or 5 further comprising means for comparing a carry value produced at the circuit's output with an earlier-generated carry look-ahead value, to determine whether the carry value could be improved.

7. An arrangement (100) for LZ1 compression of a data string, the arrangement comprising:

an input buffer (140) for holding a first sequence of bytes of the data string;

a history buffer (110, 120) for holding a second sequence of bytes of the data string;

a comparison matrix (170) coupled to the input buffer and the history buffer and having a plurality of rows and columns of comparison units (400) for comparing bytes held in the input buffer with bytes held in the history buffer, bytes of the history buffer being coupled to diagonally displaced comparison units in the comparison matrix;

detecting means (150) for detecting in each of the rows the column in which a largest number of consecutive byte matches has occurred at the comparison unit in that row and preceding comparison units in the same column; and

means (160) for encoding as a token a sequence of matched bytes detected by the detecting means,

the comparison units comprising circuits (640) according to any one of claims 1-6.

8. A method for use in a comparison matrix (170) for LZ1 compression of a data string by comparing bytes held in an input buffer (140) with bytes held in a history buffer (110, 120), the method comprising:

providing a plurality of logic gate stages (720, 730, 740, 750) coupled in series, each of the plurality of logic gate stages producing a carry value which is passed to one of the output of the unit and another of the plurality of logic gate stages, the product of the number stages in the plurality of logic gate stages and the number logic gates in each of the plurality of logic gate stages being less than the number of logic gates required for an equivalent circuit having a single logic circuit stage.

9. The method of claim 8 further comprising providing an input latch (710) coupled to the input of the unit and receiving thereat a carry value from another carry-value-producing unit.

10. The method of claim 8 or 9 further comprising providing means (770) coupled to a predetermined number of the plurality of logic gate stages producing a look-ahead carry value.

11. The method of claim 10 further comprising providing an intermediate latch (760) coupled to, and receiving the carry value from, a logic gate stage coupled to the means (770) for producing a look-ahead carry value.

12. The method of claim 10 or 11 wherein the step of producing (770) a look-ahead carry value comprises providing logic means (810) coupled to outputs of comparators of the predetermined number of the plurality of logic gate stages and coupled to comparison units of other columns of the comparison matrix.

13. The method of claim 10, 11 or 12 further comprising comparing a carry value produced at the unit's output with an earlier-generated carry look-ahead value, to determine whether the carry value could be improved.

14. A method (300) for LZ1 compression of a data string, the method comprising:

holding in an input buffer (140) a first sequence of bytes of the data string;

holding in a history buffer (110, 120) a second sequence of bytes of the data string;

comparing (170), in a comparison matrix coupled to the input buffer and the history buffer and having a plurality of rows and columns of comparison units (400), bytes held in the input buffer with bytes held in the history buffer, bytes of the history buffer being coupled to diagonally displaced comparison units in the matrix comparison means;

detecting (150) in each of the rows the column in which a largest number of consecutive byte matches has occurred at the comparison unit in that row and preceding comparison units in the same column; and

encoding (160) as a token a sequence of matched bytes detected in the step of detecting (150),

the step of comparing (170) comprising performing the method of any one of claims 8-13.

15. An integrated circuit comprising the circuit of any one of claims 1-6 or the arrangement of claim 7.

16. The integrated circuit of claim 15 wherein the integrated circuit is an application specific integrated circuit (ASIC).

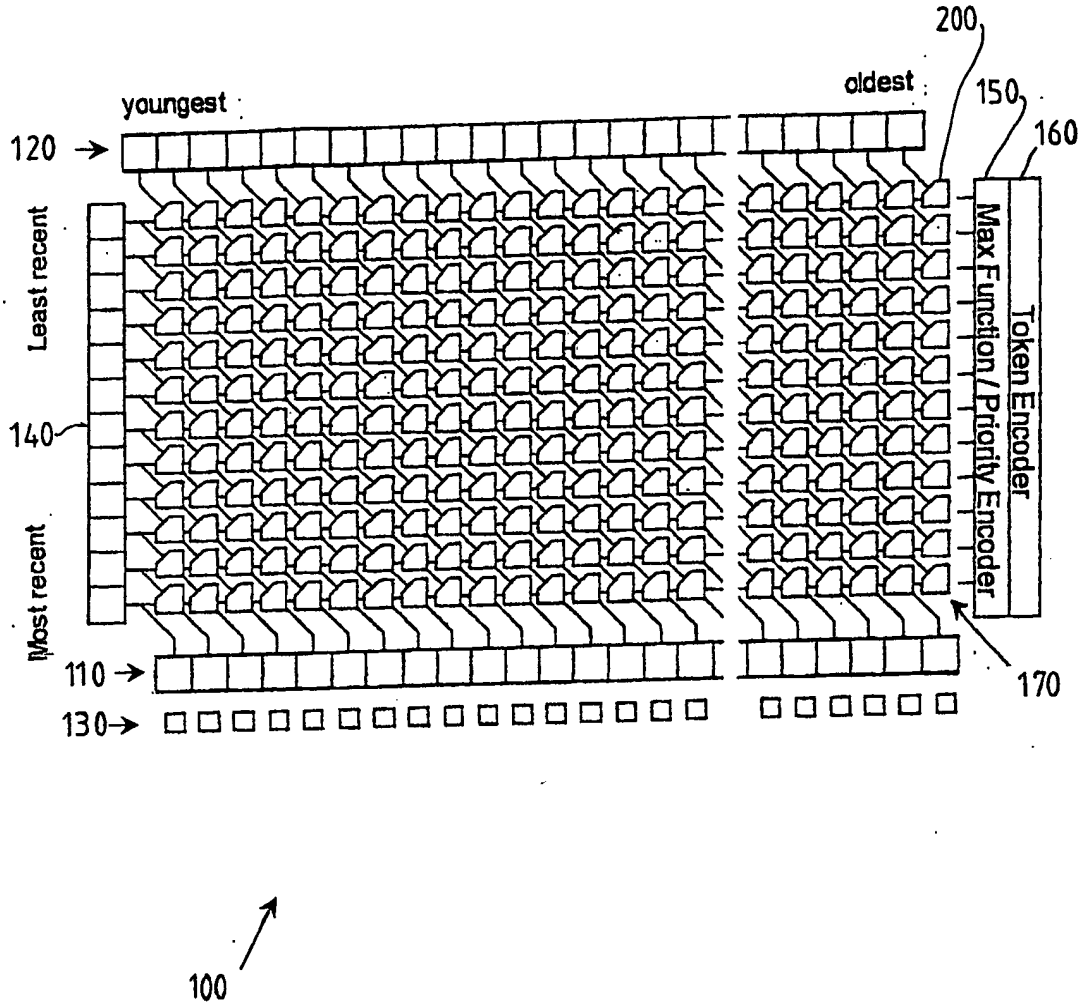

200
oldest
youngest
150
160
120
Least recent
Max Function / Priority Encoder
Token Encoder
140
Most recent
110
170
130
100

FIG. 1

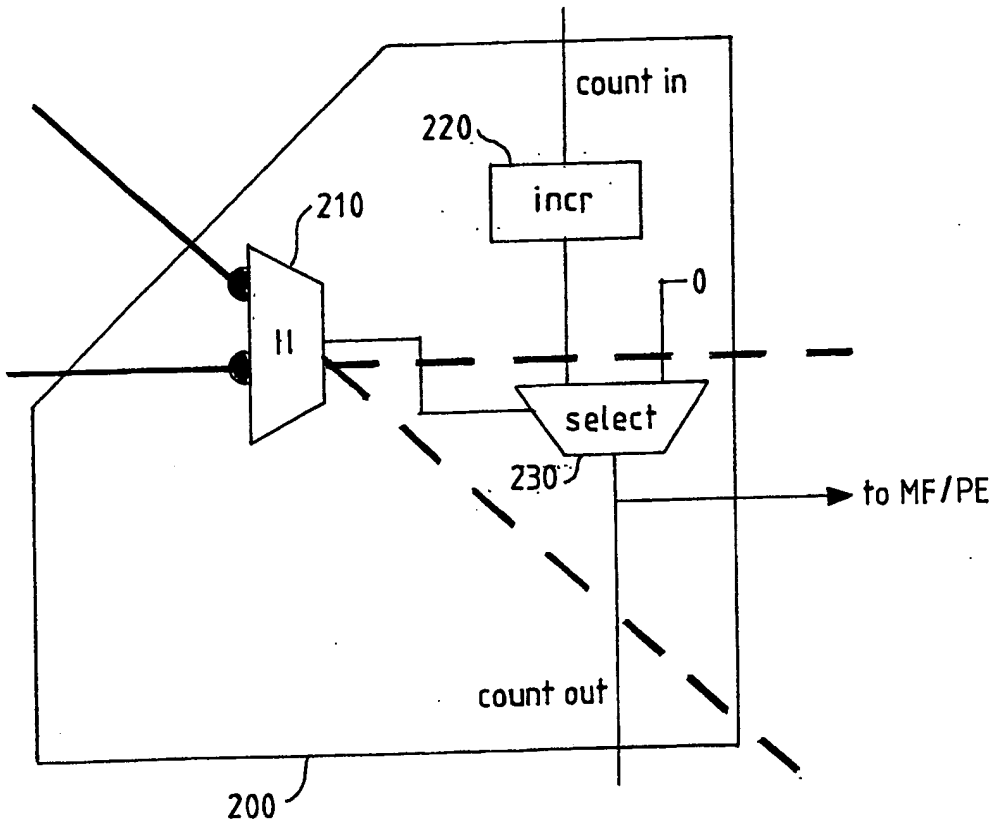


FIG. 2

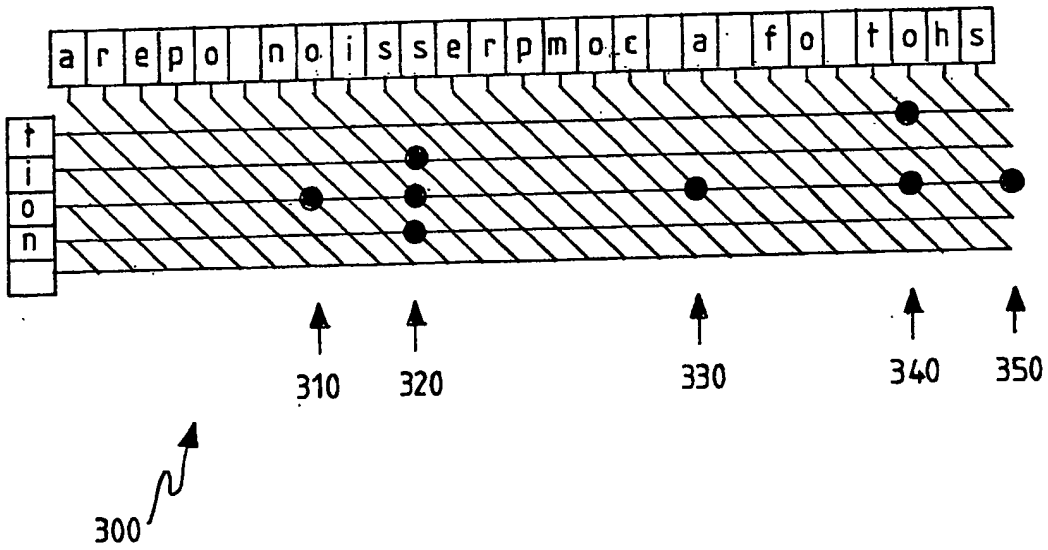


FIG. 3

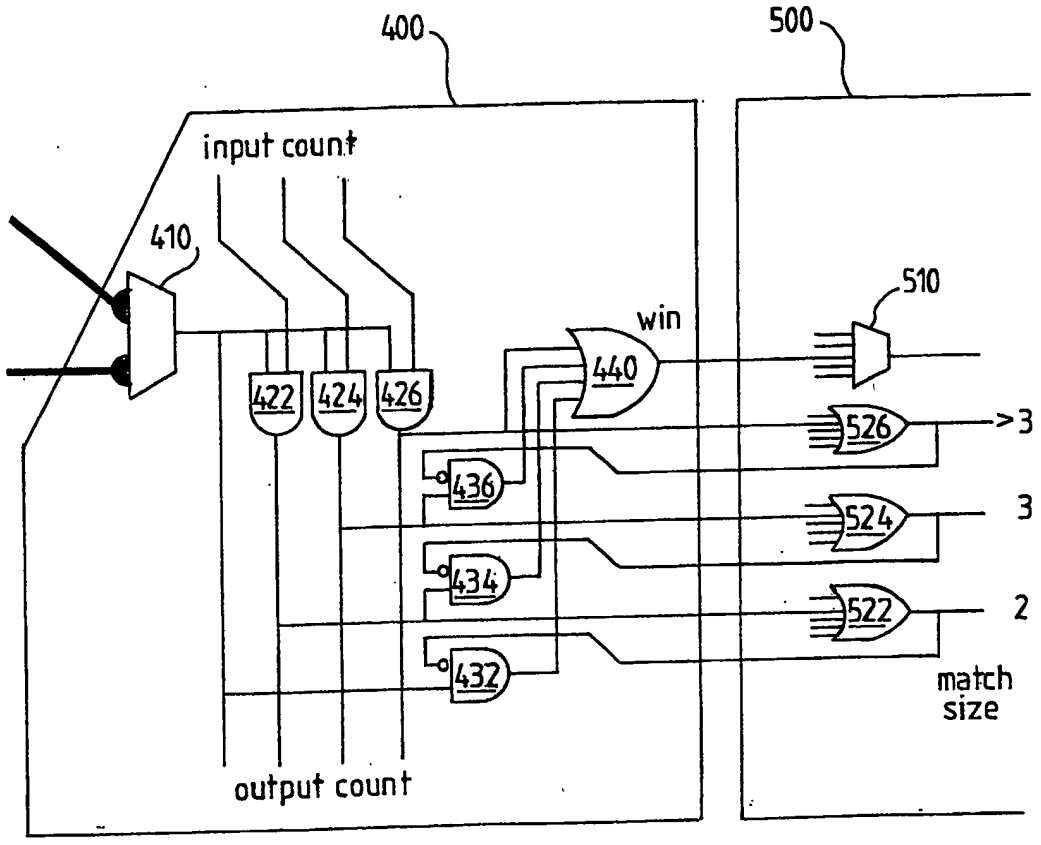

400
500
input count
410
422
424
426
win
440
510
526
>3
436
524
3
434
522
2
432
match
size
output count

FIG. 4

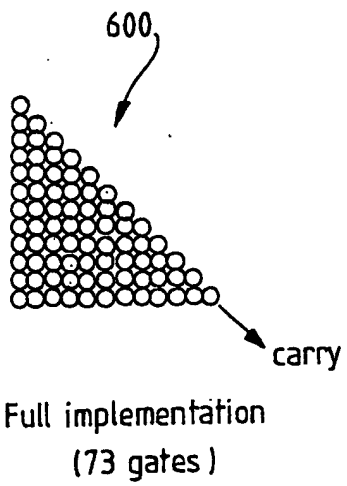

600
carry
Full implementation
(73 gates)

FIG. 5A

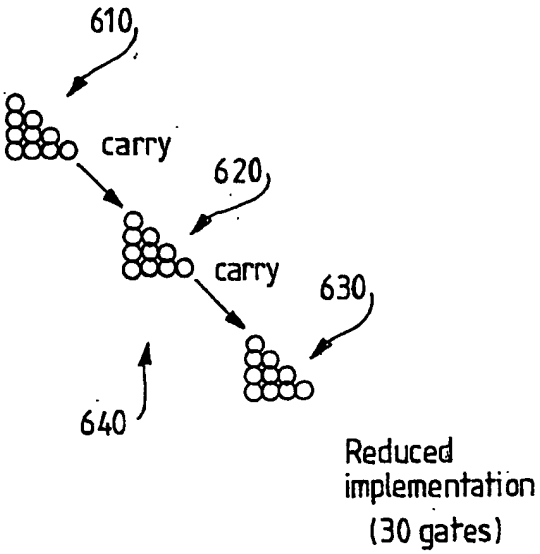

610
carry
620
carry
630
640
Reduced
implementation
(30 gates)

FIG. 5B

FIG. 6A

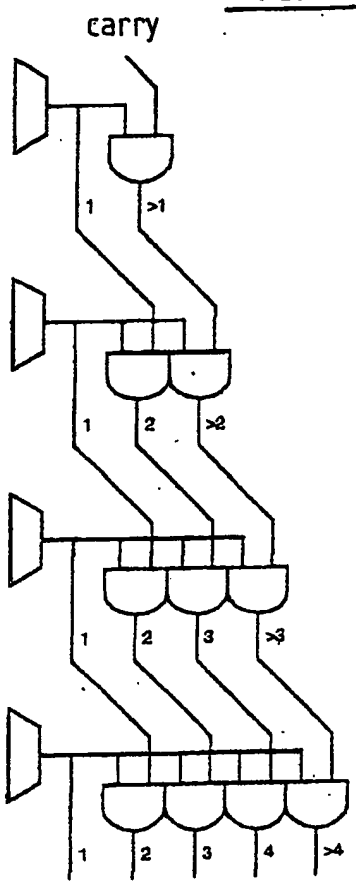
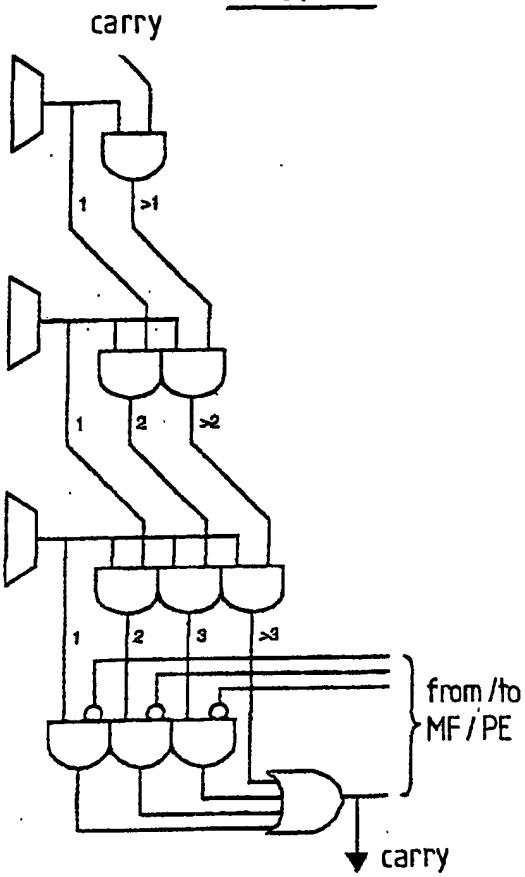
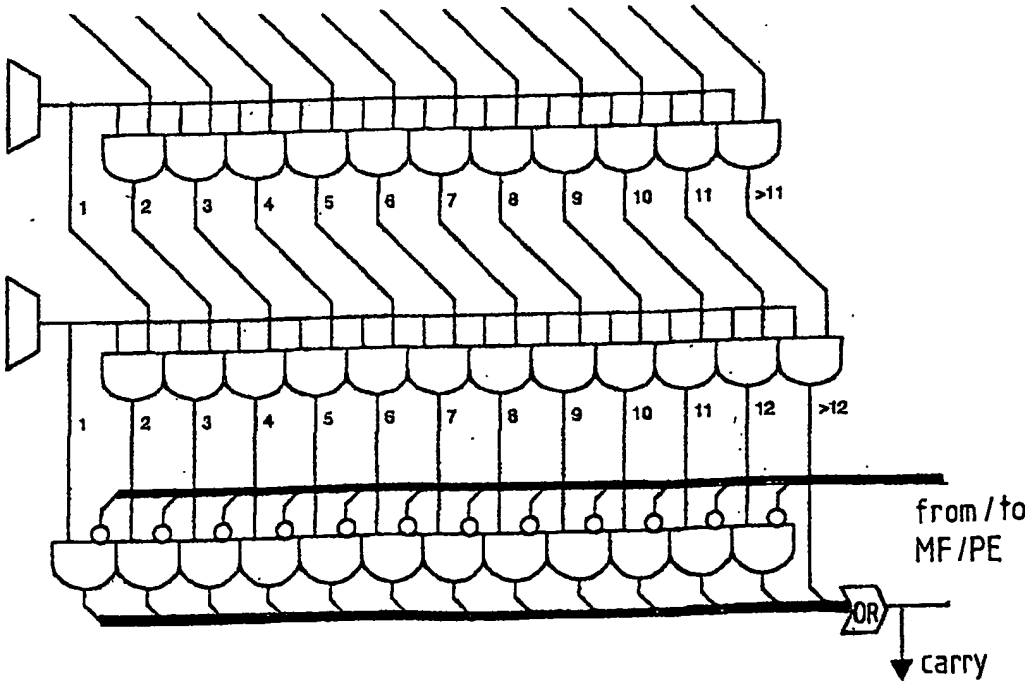

carry
1
>1
1
2
>2
1
2
3
>3
1
2
3
4
>4

FIG. 6B


carry
1
>1
1
2
>2
1
2
3
>3
from /to
MF / PE
carry


1
2
3
4
5
6
7
8
9
10
11
>11
1
2
3
4
5
6
7
8
9
10
11
12
>12
from / to
MF/PE
OR
carry

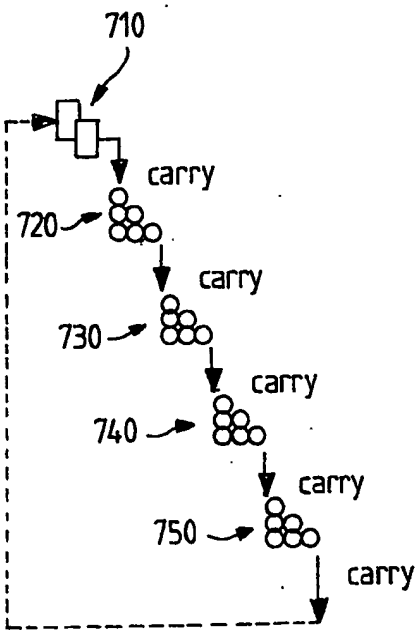

710
carry
720
carry
730
carry
740
carry
750
carry

FIG. 7A

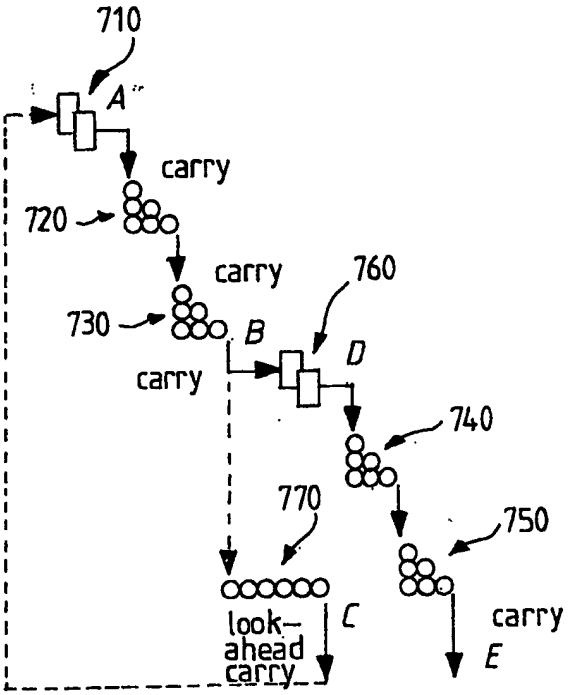

710
A
carry
720
carry
760
730
B
D
carry
740
770
750
look-
ahead
carry
C
carry
E

FIG. 7B

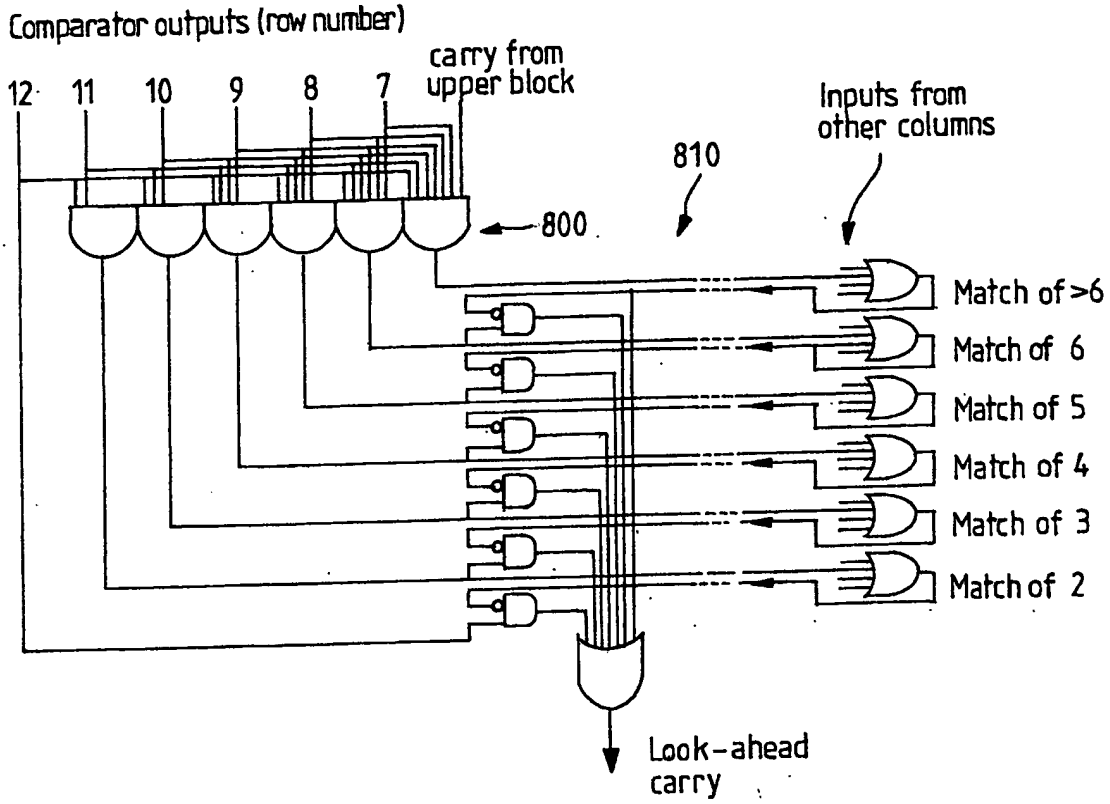

Comparator outputs (row number)
12
11
10
9
8
7
carry from upper block
800
810
Inputs from other columns
Match of >6
Match of 6
Match of 5
Match of 4
Match of 3
Match of 2
Look-ahead carry

FIG. 8

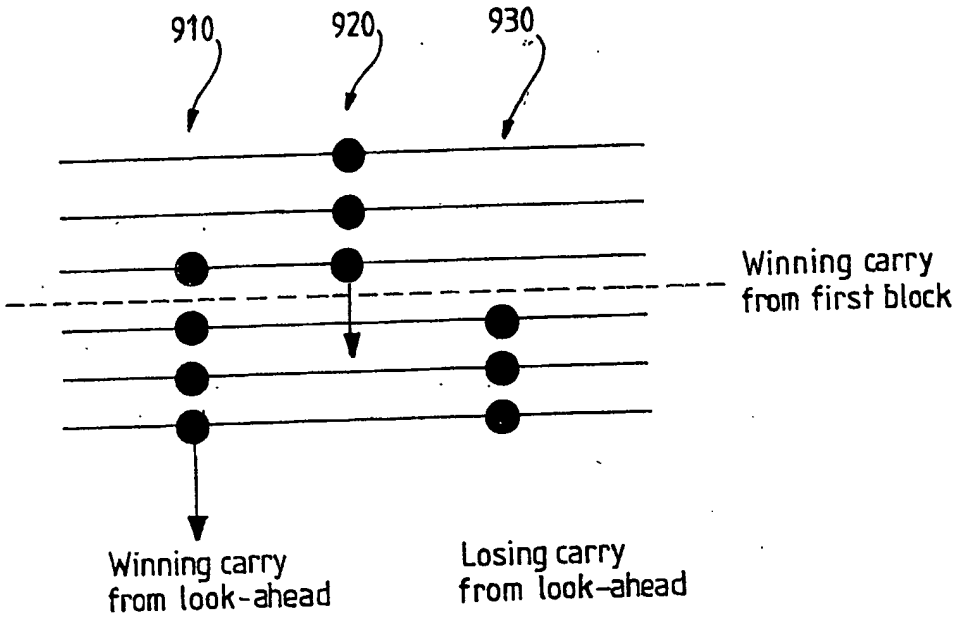

910
920
930
Winning carry
from first block
Winning carry
from look-ahead
Losing carry
from look-ahead

FIG. 9

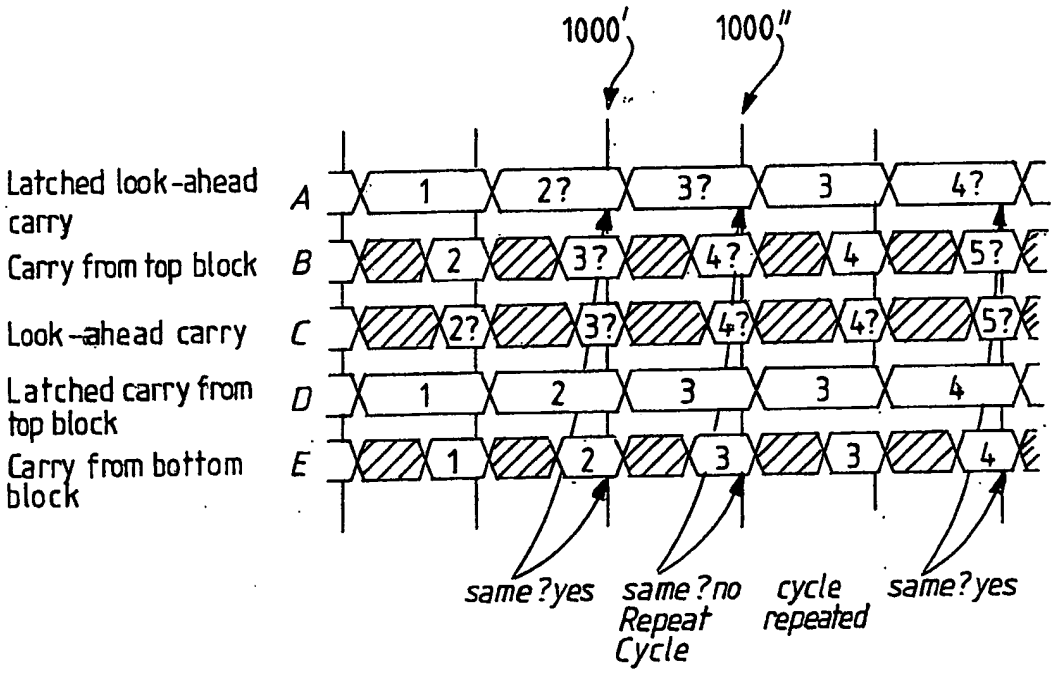

1000'
1000"
Latched look-ahead carry
A
Carry from top block
B
Look-ahead carry
C
Latched carry from top block
D
Carry from bottom block
E
1
2?
3?
3
4?
2
3?
4?
4
5?
2?
3?
4?
4?
5?
1
2
3
3
4
1
2
3
3
4
same?yes
same?no Repeat Cycle
cycle repeated
same?yes

FIG. 10